DEGREE OF MASTER OF SCIENCE

MSc in Computer Science

Database Systems Implementation

Hilary Term 2016

Your answer to this paper is to be handed in at the Examination Schools, High Street, by 12 noon on 18th April 2016

The assignment should be submitted in an envelope, clearly marked with your candidate number, (but not your name) and the name of the course, and addressed to the Chairman of Examiners, MSc in Computer Science.

NB: You must NOT discuss this examination paper with anyone.

This exam paper consists of three questions, you must answer all questions. There is a total of 100 marks available for this paper. The produced code and benchmarking results are to be submitted on paper and CD.

TURN OVER

This exam paper consists of three questions, you must answer all questions. There is a total of 100 marks available for this paper. The produced code and benchmarking results are to be submitted on paper and CD.

Your task is to design, implement, and benchmark `IncDB`, an efficient solution for maintenance of query results under updates to the input database.

A simple approach to this problem is given by the following naïve solution, which we call `NaïveDB`: Given a query and a database, `NaïveDB` computes the query result; on updates to the database, `NaïveDB` computes the new query result from scratch.

In contrast, `IncDB` avoids recomputation from scratch whenever possible. Its underlying idea is to materialize the result of the query on the initial database and then for subsequent updates to propagate the changes to the input database through the query to its result with minimal local adjustments. This approach is called *incremental maintenance* in the literature. It may require much less effort than the brufe-force `NaïveDB`, which recomputes the entire query result for each update of the database.

You will consider two types of updates:

- `+R(a,b)` states the insertion of record `(a,b)` in table `R`. In case the record is already in `R`, then the update triggers no change to `R`. Otherwise, the record is added to `R`.
- `-R(a,b)` states the deletion of record `(a,b)` from table `R`. In case the record is not in `R`, then the update triggers no change to `R`. Otherwise, the record is deleted from `R`.

You will also consider two query languages:

- **Join queries with equality conditions, henceforth called equi-join queries**.

  They have the general form $R_1(\bar{A}_1), \ldots, R_n(\bar{A}_n)$, where $R_1, \ldots, R_n$ are relation symbols, $\bar{A}_1, \ldots, \bar{A}_n$ are lists of query variables, and the comma means conjunction. Joins are expressed by using the same variable in several of these lists. Examples:

  - $R_1(A, B), R_2(A, C)$ joins two tables R_1 and R_2 on column A.
  - $R_1(A, B), R_2(A, C), R_3(B, C)$ is the triangle query that joins any two of the three tables on one of their columns.

- **Count aggregates on top of equi-joins, henceforth called count queries**.

  They have the general form COUNT$[Q]$, where Q is an equi-join query. Examples:

  - COUNT$[R_1(A, B), R_2(A, C)]$ computes the cardinality of the equi-join query $R_1(A, B), R_2(A, C)$, i.e., it computes the number of distinct assignments (a, b, c) to the variables (A, B, C).
  - COUNT$[R_1(A, B), R_2(A, C), R_3(B, C)]$ computes the cardinality of the triangle query $R_1(A, B), R_2(A, C), R_3(B, C)$, i.e., it computes the number of distinct triangles.

Note on Questions 1 and 2: Each of these questions has several parts of increasing complexity. Part (a) of Question 1 asks for the design of `IncDB` for two simple queries under inserts and

deletes, while Part (a) of Question 2 asks for the implementation of this limited version of `IncDB`. Similarly, Parts (b) of these two questions refer to more complex query languages. It is advisable to address this exam paper layer by layer so as to maximize the number of marks you can get.

You may assume the following simplifications: The tables are sets of distinct tuples, i.e., the tables do not accommodate duplicates. Furthermore, each table has two columns.

Question 1

This question has three parts and is worth 40 = 15+15+10 marks.

Your first task is to **design** `IncDB` that provides efficient maintenance of query results. In case your design is inspired by existing research literature, you must appropriately cite the source and explain in detail the distinct novel aspects of your approach.

(a) First consider the equi-join query $R_1(A, B), R_2(A, C)$ and its counting version $\text{COUNT}[R_1(A, B), R_2(A, C)]$ over *arbitrary databases* with tables R_1 and R_2.

(i) Give in-memory algorithms in pseudocode for the incremental maintenance of the results of each of these queries under a sequence of inserts and deletes of records in the input tables R_1 and R_2.

(ii) For each of the two algorithms, give a complete example showing how the algorithm works. This includes: the input database of your choice, the query result, how the algorithm updates the query result under one insertion of a new record to one of the tables and then one deletion of an existing record distinct from the previously inserted record.

(iii) Explain why your algorithms are superior to `NaïveDB` for workloads consisting of one update at a time or of bulk updates. Your explanation may refer to the computational complexity of each of the two approaches or any other measure of performance that you find appropriate.

(iv) Describe and motivate the data structures used by your algorithms for representing the input tables and the query results.

Hint: When inserting a record t into table R_1, the new query result $(R_1 \cup \{t\}) \bowtie R_2$ can be rephrased as $R_1 \bowtie R_2 \cup \{t\} \bowtie R_2$, where $R_1 \bowtie R_2$ is the original query result. (15 marks)

(b) Consider now *arbitrary equi-join queries over arbitrary databases* and address the same questions (i) to (iv) from Part (a): present an incremental maintenance algorithm for equi-join queries; give a complete running example; explain why your algorithm is better than `NaïveDB`; and describe the data structures used by your algorithm. The performance explanation and the data structures may be the same as for Part (a), though generalised to arbitrary equi-join queries. (15 marks)

(c) Consider now *arbitrary count queries over arbitrary databases* and address the same questions (i) to (iv) from Part (a): present an in-memory incremental maintenance algorithm for count queries; give a complete running example; explain why your algorithm is better than `NaïveDB`, and describe and motivate the data structures used by your algorithm. The performance explanation and the data structures may be the same as for Part (a), though generalised to arbitrary count queries. (10 marks)

Question 2

This question has three parts and is worth 40 = 15+15+10 marks.

Your second task is to **implement** `IncDB`. Your implementation must follow your design proposed for Question 1. In addition to the general simplifications (tables are binary and represent sets of records), you may also assume the following simplifications:

- All tables can be kept in main memory and their columns are of type integer.
- All updates are serial, so no transaction or locking mechanism is needed.
- There is no need to implement a query parser, the queries can be constructed in your code.

You must state explicitly which of these and possibly further simplifications you assumed.

For this implementation task, you may use a programming language of your choice as long as the code is compilable and runnable on the lab machines. If needed, you may also use open-source implementations of standard data structures, e.g., tries or hash maps, and of standard algorithms presented in the course for sorting, indexing, and joining.

Your implementation must be accompanied by documentation on how to compile, run, specify new queries and updates in the code, and benchmark your `IncDB`. The code must also be sufficiently well commented. Marks will be deducted if your code is not compilable, runnable, or commented or if the documentation is missing.

(a) Implement your incremental maintenance algorithms from Question 1 Part (a) for the two simple queries. (15 marks)

(b) Implement your incremental maintenance algorithm from Question 1 Part (b) for arbitrary equi-join queries. (15 marks)

(c) Implement your incremental maintenance algorithm from Question 1 Part (c) for arbitrary count queries. (10 marks)

Question 3

This question has three parts and is worth 20 = 8+12 marks.

Your third task is to **benchmark** `IncDB` against `NaïveDB`. `NaïveDB` may be implemented using hand-crafted join plans for the specific queries in Parts (a) and (b) below and that use your sort-merge join implementation from the lab assignment.

For a fair comparison, the input database has identical representation for both approaches. You must document how to run your experiments and provide a script to run them and generate the plots or the values to be plotted. You must explain each plot included in your answer – this means explaining why `IncDB` and `NaïveDB` behave the way they do in your experiments. Marks will be deducted if the documentation, the script for running the experiments, or the explanation of experimental findings are not satisfactory.

Consider the four binary tables $R_1(A, B)$, $R_2(A, C)$, $R_3(B, C)$, and $R_4(A, D)$ and ten sets of update statements that can be found on the course web page[1] as CSV files with one record or

[1] https://www.cs.ox.ac.uk/teaching/materials15-16/databasesystemsimplementation/exam-data.zip

update per line. For instance, for table R_1, the record $R_1(1,1)$ is stored as one line with content `1,1` in the file `R1`. For insert set I_1, the insert statement `+R1(1,1)` is stored as one line with content `1,1,1` in the file `I1`. For delete set D_1, the delete statement `-R1(1,1)` is stored as one line with content `1,1,1` in the file `D1`. They were generated as follows:

- Table R_i has 10^{i+1} distinct records, $\forall 1 \leq i \leq 4$. The values for A, B, C, D are drawn at random from the range $[0, 10^3]$.
- There are five sets of insert statements $I_1, \ldots, I_5$, where the set I_j has 10^j inserts `+R`$_i$`(x,y)` generated as follows. The values i, x, and y are drawn at random from the ranges $[1, 4]$, $[0, 10^3]$, and respectively $[500, 10^4]$.
- There are five sets of delete statements $D_1, \ldots, D_5$, where the set D_j has 10^j deletes `-R`$_k$`(x,y)` generated as follows. The value k is drawn from the range $[1, 4]$ with probabilities $P[k = 1] = 5 \cdot 10^{-4}$, $P[k = 2] = 5 \cdot 10^{-3}$, $P[k = 3] = 5 \cdot 10^{-2}$, and $P[k = 4] = 1 - \sum_{1 \leq i \leq 3} P[k = i]$. The set D_j contains $n_1 + \ldots + n_5 = 10^j$ delete statements, where n_i is number of times k is chosen with value i. Then, these n_i delete statements correspond to deleting n_i records from table R_i.

(a) For each of the two queries from Question 1 Part (a), draw the following two plots. The first plot has on the x-axis the sets of inserts $(I_1, \ldots, I_5)$ in increasing order of their sizes, and on the y-axis the performance of `IncDB` in wall-clock seconds (user time) to execute each set I_j of inserts. The second plot has on the x-axis the sets of deletes $(D_1, \ldots, D_5)$ in increasing order of their sizes, and on the y-axis the wall-clock performance of `IncDB` in seconds to execute each set D_j of deletes.

You may choose to execute the updates in sequence or in bulk. The time to compute the query result on the initial database before updates should not be included in the plots. The reported times should not include the times to execute the inserts or deletes on the database and should only reflect the time to propagate these changes to the query result.

Add the performance for recomputing the query result from scratch after each update using `NaïveDB` to the plots. As for `IncDB`, you should only report the times taken to compute the query result after each set of updates and without the time taken to compute the original result and the actual database updates.

You should run each experiment five times and plot the mean time performance (optionally with error bars). You may decide on a reasonable timeout per experimental setting (say, one hour) in case one of the two query engines takes too long. (8 marks)

(b) Perform the experiment from Part (a) for the following queries:

(i) The equi-join query $R_1(A, B), R_2(A, C), R_4(A, D)$ and its counting version.

(ii) The equi-join query $R_1(A, B), R_2(A, C), R_3(B, C)$ and its counting version.

(12 marks)

LAST PAGE